LIBRARY OF THE
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

510.84
Il6r
no. 337-342
cop. 2

Digitized by the Internet Archive
in 2013

http://archive.org/details/modelforlinearpr340gold

IIR6N
no. 340

Report No. 340

Math

A MODEL FOR LINEAR PROGRAMMING OPTIMIZATION OF I/O-BOUND PROGRAMS

by

David E. Gold

June, 1969

THE LIBRARY OF THE
AUG 25 1969
UNIVERSITY OF ILLINOIS

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. 340

A MODEL FOR LINEAR PROGRAMMING OPTIMIZATION OF I/O-BOUND PROGRAMS

by

David E. Gold

June, 1969

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

* This work was supported in part by the Advanced Research Projects Agency as administered by the Rome Air Development Center under Contract No. US AF 30(602)4144 and submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science, June, 1969.

A Model for Linear Programming Optimization of I/O-Bound Programs

ABSTRACT

A model of a class of machines having periodically addressable secondary memory is derived. The problem of (timewise) optimization of programs consisting of many core-loads is considered with respect to the model. For a large class of programs, sufficient constraints are established to make the solution amenable to linear programming methods.

ACKNOWLEDGMENT

I wish to thank Professor D. J. Kuck for his suggestions and criticisms during the evolution of this study.

I would also like to extend my thanks to the Department of Computer Science and the ILLIAC IV Project for their financial support.

Finally, I am grateful to Mrs. Sharon Hardman for her excellent job in preparing this manuscript.

TABLE OF CONTENTS

		Page
1.	INTRODUCTION	1
	1.1 History	1
	1.2 Description of the Algorithm	2
	1.3 Evaluation	3
2.	PRELIMINARY	4
	2.1 Code Graph	4
	2.2 Basic Assumptions	6
3.	SIMPLE COMPUTATION MODEL AND ESTABLISHMENT OF CONSTRAINTS	9
	3.1 Basic Model	9
	3.2 Memory Constraint	11
	3.3 Disk Timing Constraints	12
	3.4 Additional I/O Constraints	13
	3.5 Sequencing Constraints	14
	3.5.1 Standard	14
	3.5.2 Special Case: Temporary Variables	17
	3.6 I/O Contiguity Constraints	18
	3.7 System Constraints	19
	3.8 Criterion Function	20
4.	BOUNDARY CONDITIONS	21
	4.1 Constraints Which Apply to All Nodes	21
	4.2 New Sequencing Constraints	22
5.	OTHER CONSIDERATIONS	24
	5.1 Real Code	24
	5.1.1 Looping	24
	5.1.2 Branching	25

Page

5.2 Overlapping Nodes. 27
5.3 Reduction of Variables 27

APPENDIX

A. SUMMARY OF CONSTRAINTS. 30

B. EXAMPLE 32

1. INTRODUCTION

1.1 History

As computers become faster and larger, programs which run on them expand also. For an increasing proportion of programs (those manipulating very large amounts of data), a large amount of the "execution time" may not be spent executing. The speed at which these programs run is a function of the time it takes to manipulate their necessary data between primary and secondary memory. It is this class of problems (commonly called "I/O bound") to which the method to be described is directed. The objective was to develop a model of a program and a model of both primary and secondary memories of the processor on which it is to be executed. The model must be suitable for analysis. In particular, the goal was to formulate a linear programming specification of the model such that, for most types of source code, the solution is an altered code which completes its execution in the shortest possible time and is computationally equivalent to the original program.

It is obvious, however, that there can exist no algorithm for reading programmer's minds. This immediately precludes consideration of every program which is computationally equivalent to the original. On the other hand, a very large subset of these is obtained from consideration of the original code and permutation of its statements. Indeed, it is the reordering of statements alone which may allow for the transferring of data at optimal or near optimal times.

1.2 Description of the Algorithm

The algorithm presented here was developed for use on a machine (or, more precisely, system) having one or more fixed head disks. The method is, in fact, sufficiently general to handle any system with periodically accessible secondary memory (e.g., drums, delay lines). For obvious historical reasons, the word "disk" as used in this paper is synonymous with the more general secondary memory devices.

The algorithm first models a program--in sections--with a directed graph (called a code graph). This code graph represents the structure of the replacement statements in each section of code such that reorderings of these statements, when beneficial for I/O reasons, are guaranteed not to affect the final calculated results of the code.

The execution of the program being modeled is quantized in the model. That is, the time axis is considered to be a series of small increments of time in which various operations (execution, input, output, etc.) may occur. Since there is no a priori knowledge of the optimal solution, all such operations are initially allowed. This is accomplished by considering a large number of binary integer variables as "flags" or indicators. If a particular variable is assigned the value "1" in a solution, the operation with which it is associated is performed in the time interval to which it corresponds.

It is upon this set of variables that the constraints are placed, guaranteeing a solution which is physically realizable but not restrictive with respect to unusual use of the I/O facilities. Such techniques as multiple copies of data on the disk and relocating data on the disk are therefore allowed in the final solution.

1.3 Evaluation

While the model presented in this paper yields too many variables to be adequately tested with present linear programming methods, several observations may nonetheless be made.

There will be an obvious trade-off between the time taken to optimize a program (the execution time for an LP program to find a solution for the model derived here) and the time which it actually saves. At one end of the spectrum are those programs which spend a relatively small part of their time waiting for I/O. At the other end are programs in which the majority of the execution time is really I/O "wait" time. It is clearly towards the latter direction that the model presented in this paper will be useful. At just what point in the spectrum it will begin to become economical, only testing with one or more LP programs can determine. [It should also be mentioned that the ability to optimize code is highly dependent upon the freedom in reordering statements of the code. In cases where little or no reordering is possible, while the model will yield constraints forcing a rapid solution, little optimization may be accomplished.]

2. PRELIMINARY

2.1 Code Graph

A code graph corresponding to a set of instructions is defined by the following algorithm.

The code must have no transfers (i.e., is assumed to be a sequential list of operations).

1) For each variable* name which has a second occurrence on the left of a replacement statement (that is, any variable which has its value assigned more than once in the code), do the following:
   a) rename the second such occurrence such that the new name is unique with respect to the variables already appearing in the code.
   b) rename all occurrences of the same variable which occur subsequent to the one changed to the same name (this is done on the left side of replacement statements, if necessary, yielding new "second occurrences").
2) repeat 1 until no more such occurrences remain.
3) obtain a graph from this modified code by placing nodes in one-to-one correspondence with the replacement statements in the code. Each node is to be connected (by directed branches) to those nodes corresponding to the computation

---

*By "variable" here is meant the standard programming concept of variable as applied to large blocks of data (e.g., arrays, matrices).

of variables which appear on the right side of the statement to which it corresponds. The direction assigned to each branch is from predecessor to successor (intuitively, the order in which the computation must take place).

Initial (non-computed) variables have their corresponding nodes placed such that they are not successors of any nodes.

Example:

Consider the following segment of code (where no assumptions are made about the meaning of the operations +, -, *):

A←B*C

D←B+A-C

A←A-D

A←A*D

Step 1, first iteration yields:

A←B*C

D←B+A-C

A'←A-D

A'←A*D

Step 1, second iteration

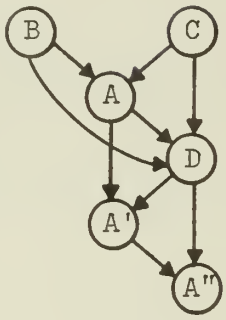
A←B*C

D←B+A-C

A'←A-D

A"←A'*D

The graph corresponding to this is:



With each node on this code graph is associated the execution time for the operation which corresponds to that node. It is assumed that all operations (for given data sizes) have a known execution time; this is then normalized so that it is represented as an integer multiple of the time intervals (to be defined later) chosen for the particular model.* This normalized time is denoted q_i for the i-th node.

Note that a node may be considered as both an operation (calculation) on some data (which we assume to take a fixed amount of time) as well as the generation of some data. For ease of explanation, then, the word "node" will be used interchangeably for both calculations and produced data, where such use is unambiguous.

2.2 Basic Assumptions

The following simplifying assumptions are made. All calculated data is to be saved. (That is, no provision is made for recognizing purely

---

*That is, the time intervals are chosen to be at least as short as the shortest execution time so that all execution time may be represented as multiples of time intervals.

temporary intermediate results, so one copy of everything will eventually be stored on the disk. (It will be shown later that the model can be refined to avoid this inefficiency.)

The disk is assumed to be infinite. In other words, the amount of data to be written on the disk is significantly smaller than the disk's capacity; hence, the assumption that disk space is always available when and where it is required.

The actual transmission time for each block is negligible compared to the rotation (cycle) time of primary memory. Primary memory, while fixed in size, is assumed to be capable of storing (holding) data in a form suitable for direct calculation. It may, however, be the case that for many programs it is not possible to use primary memory with perfect efficiency. This is somewhat analogous to having a number of differently dimensioned rectangles and attempting to place them (non-overlapping) in a larger rectangle, the area of which is equal to the combined areas of the smaller rectangle. The situation occurs here because the model does not consider formatting of primary memory whereas, for large problems of the type being considered, its contents (data) must be highly structured.

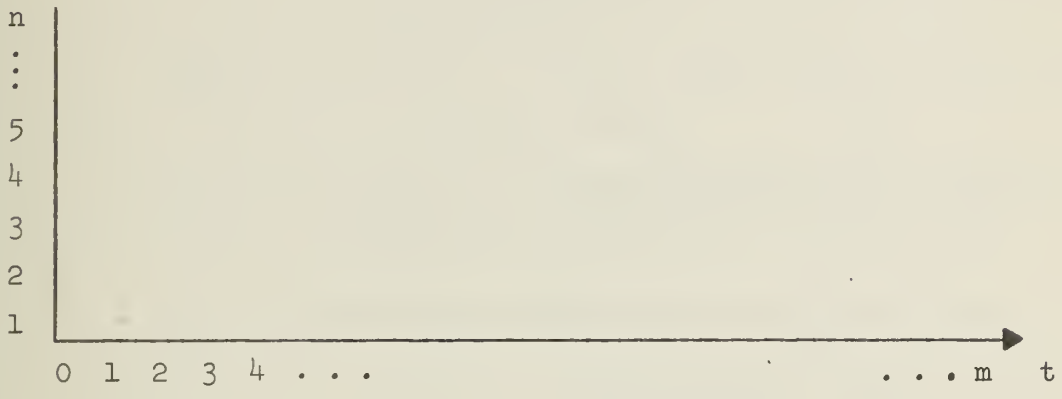
A reasonable approximation to real life can be obtained here by considering primary memory as being smaller than it actually is. That is, a certain "bias" is introduced to account for the "holes" or unused primary memory due to the rigid data structure. This bias is best considered to be particular to each program and memory size. For the remainder of this paper, $Memory_{max}$ or Mem_{max} will be considered to be the biased maximum useable primary memory. In other words, no storage formatting problems are considered. (Thus, it is possible to effect legitimate solutions to optimizing

code by making the memory "look" smaller than it really is for the machine under consideration. This has the effect of allowing enough extra room to store arrays which do not naturally "fit together" in the restricted memory.)

3. SIMPLE COMPUTATION MODEL AND ESTABLISHMENT OF CONSTRAINTS

3.1 Basic Model

The model is a two-dimensional array of vectors. The array is as illustrated below:



where time is the abcissa. This means that each column refers to a unique time interval. The intervals are of equal length, their minimum (time) is such that every calculation time may be expressed an integral number of intervals. m is the total number of such intervals and is fixed by obtaining any upper bound on computation time for the entire section of code being considered.

Clearly, an upper bound on the running time is easily obtained.* The time interval and m determine the amount of time we consider--not the time the program runs. Indeed the program should execute in somewhat less time than this upper bound, the optimization criterion being that the last

---

*A perfectly legitimate upper bound of total computation time may be obtained by just considering the original code-executed in sequential order with appropriate delays for I/O.

block to finish computing do so at the earliest possible time (used number of time intervals is minimum).

The ordinate in the matrix is node number (the nodes having been ordered arbitrarily).

Each entry in the matrix is a vector having six elements. Thus, we have the following set of variables:

$$x_{ijk}$$

where k is the time interval $0 \leq k \leq m$

where j is the node number $1 \leq j \leq n$

and $1 \leq i \leq 6$ corresponding to the actions

- i = 1 computation
- i = 2 input (begin)
- i = 3 input (continuing)
- i = 4 output (begin)
- i = 5 output (continuing)
- i = 6 overlay

Recalling the concept of equivalence between node, computation and data, the meaning of the variables x_{ijk} is as follows:

$x_{1jk} = 1$ iff the j-th node is "computing" in the k-th time interval

$x_{2jk} = 1$ iff the j-th node (data) starts being input in the k-th time interval (that is, this data was not being input during the k-1th time interval)

$x_{3jk} = 1$ iff the j-th node is being input in the k-th time interval

$x_{4jk} = 1$ iff the j-th node starts being output in the k-th time interval

$x_{5jk} = 1$ iff the j-th node is being output in the k-th time interval

$x_{6jk} = 1$ iff the memory space occupied by the j-th node is overlaid (released, made available) immediately at the beginning of the k-th time interval

In the part that follows, one or more of the subscripts may not appear when our concern is only with the remaining one(s) and the meaning is clear. The various constraints on the variables will now be introduced.

3.2 Memory Constraint

The amount of available primary memory is fixed and the amount of data contained therein must not exceed its capacity. This yields m constraints:

$$\sum_{r=0}^{k} \sum_{j=1}^{n} \left(\frac{x_{3jr}}{p_j} + \frac{x_{1jr}}{n_{c_j}} - x_{6jr} \right) \cdot (\text{size of the j-th block}) \leq \text{Memory}_{max}$$

The above inequality must hold for all values of k, $1 \leq k \leq m$, yielding m constraints. p_j is the number of time intervals required to transmit the j-th block (of data). The summation, then, yields 0 iff the j-th block is not in primary memory at time k (more accurately: at the k-th interval); otherwise, the result is the sum of all memory taken up by blocks in memory plus the fractional portion of any blocks in the process of being input. The model thus assumes that blocks of data are input in such a fashion that the amount of memory taken up by them is simply a ramp function over the time intervals in which they are being input.

Note that it is possible to substitute the simpler constraint

$$\sum_{r=0}^{k} \sum_{j=1}^{n} \left(x_{2jr} + \frac{x_{1jr}}{n_{c_j}} - x_{6jr}\right) (\text{size of the j-th block}) \leq \text{Memory}_{max}$$

for the previous one. The difference is in the resolution of occupied memory: the latter constraint requires that room be available in memory for an entire block when the transmission begins. This is analogous to a square wave description of memory usage.

3.3 Disk Timing Constraints

Here the fact that we are considering only fixed head disks is used. We require that the time between when a block is output and subsequently input be a multiple of the disk revolution time.* (The reason is

---

*Note that actual transmission time is assumed to be negligible. In cases where this is not the case, however, the time between output and subsequent input is an integral number of disk rotations plus the time for two transmissions of the data. The constraint is then modified accordingly.

obvious: data cannot wander around on the disk--it must remain where it is put.)

Let r be the (integral) number of time intervals corresponding to a single disk revolution. Then for each node j,

$$x_{2jk} - x_{4\,j\,k-\ell r} \leq 0 \quad \text{for some integer } \ell \geq 1$$

(for $x_{2jk} = 1$ some x_{4jq}, an integral number of revolutions ago must be 1. For $x_{2jk} = 0$, we are not concerned with the values of x_{4jq} for integral numbers of revolutions ago.) Written as a linear constraint:

$$\forall j,\ \forall k \quad x_{2jk} - \sum_{t=s}^{f,r} x_{4jt} \leq 0,\ s = k \text{ modulo } r,\ f = k-r$$

(note that r is a parameter which is constant for any particular system, allowing the generation of the above constraints.)

3.4 Additional I/O Constraints

The disk timing constraints are sufficient only for continuous data transfer. That is, they guarantee only that initiation of input and output be "modulo the disk". Here additional constraints establish continuous transmission: (let p_j be as defined previously)

$$p_j\, x_{2jk} - \sum_{\ell=k}^{k+p_j-1} x_{3\ell k} \leq 0 \qquad \forall j,\ \forall k$$

(for $x_2 = 1$ we start to input in the k-th time interval. Since there are p_j terms in the summation, there must be p_j continuous intervals of data transmission commencing with the k-th. For $x_2 = 0$, nothing is constrained.)

There is also a similar set of constraints for output:

$$p_j \, x_{4jk} - \sum_{\ell=k}^{k+p_j-1} x_{5\ell k} \leq 0$$

The above constraints do not disallow initialization of input or output more than once within any p_j intervals--which can be precluded with another set of constraints of the form

$$\sum_{\ell=k}^{k+p_j-1} x_{2j\ell} \leq 1$$

$$\sum_{\ell=k}^{k+p_j-1} x_{4j\ell} \leq 1$$

but because of the restrictions on allowable sequences of computation, input, output and overlay, these are redundant.

3.5 Sequencing Constraints

3.5.1 Standard

Let the letters C, I, O, D represent the operations (in primary memory) compute, input, output and overlay (destroy), respectively.

For any node, then, the temporal sequence of operations associated with it can be expressed

$$COO^*D(IO^*D)^*$$

where * is Kleene star. In other words, any node must first be computed, then output any number (≥ 1) times and overlaid. This may then be followed by any number (possibly zero) of the following sequences: input, output ≥ 0 times, overlaid. All non-permitted sequences are disallowed by the constraints given in this section.

$$\forall k \quad -1 \leq \sum_{\ell=1}^{k} (x_{2j\ell} - x_{6j\ell}) \leq 0$$

guarantees the proper relative sequencing of I and D, $[D(ID)^*]$.

$$\sum_{\ell=1}^{k} (x_{2j\ell} - x_{6j\ell}) - x_{4jk+1} \geq -1$$

requires that a block be (at least started to be) input before it may start being output. (Essentially, this establishes that 0's appear only in the allowed sequence.) It is still necessary to require the computation and output (in that order) before the remainder of the sequence.

Let $\overline{C}_j$ be the integer formed by considering the (binary) variables $x_{1j0}\ x_{1j1} \cdots x_{1jm}$ as an integer in binary notation.

That is, $$\overline{C}_j = \sum_{\ell=0}^{m} x_{1j\ell} \cdot 2^{(m-\ell)}$$

Similarly, $$\overline{O}_j = \sum_{\ell=0}^{m} x_{4j\ell} \cdot 2^{(m-\ell)}$$

$$\overline{D}_j = \sum_{\ell=0}^{m} x_{6j\ell} \cdot 2^{(m-\ell)}$$

then we require at least 1 overlay:

$$\forall j \qquad \sum_{\ell=1}^{m} x_{6j\ell} \geq 1$$

at least one output preceding it:

$$\overline{O} > \overline{D}_j \quad \text{or} \quad \sum_{\ell=0}^{m} (x_{4j\ell} - x_{6j\ell}) \cdot 2^{(m-\ell)} \geq 0$$

and computation preceding this:

$$\overline{C}_j > \overline{O}_j \quad \text{or} \quad \sum_{\ell=0}^{m} (x_{1j\ell} - x_{4j\ell}) \cdot 2^{(m-\ell)} \geq 0$$

The last constraint, however, is not sufficient. It is still possible for computation to be unfinished when the first output begins.

This is disallowed by forming the next constraints, viz.: let n_{c_j} be the number of time intervals for which the j-th node (computation) must take place. Then

$$\sum_{\ell=0}^{m} x_{1j\ell} = n_{c_j}$$

and it is necessary to insure that the "right-most" or latest variable with value 1 is still "to the left of" or before the first output. The constraints are

$$\forall k, \forall j \quad x_{1jk} + \sum_{\ell=0}^{k} x_{4j\ell} \leq 1 \quad \text{(for any } x_{1jk} = 1, \text{ there can be no } x_{4j\ell} = 1 \text{ for } \ell \leq k\text{)}.$$

The constraints corresponding to the inequality $\overline{C}_j > \overline{O}_j$ are included in the above set and may therefore be discarded.

3.5.2 Special Case: Temporary Variables

In the case of nodes which are only temporary variables, there is no reason to require that a copy of same be ultimately placed on the disk. The allowable sequences, then, for such nodes are CO*D(IO*D)*. Such an alteration in sequencing (no longer requiring an output) can be accomplished by substituting the inequality $\overline{C}_j > \overline{D}_j$ for the inequality $\overline{O}_j \geq \overline{D}_j$. This yields the constraint:

$$\frac{\sum_{\ell=0}^{m} x_{1j\ell} \cdot 2^{(m-\ell)}}{2^{n_{c_j}}} - \sum_{\ell=0}^{m} x_{6j\ell} \cdot 2^{(m-\ell)} \geq 0$$

or,

$$\sum_{\ell=0}^{m} \left(\frac{x_{1j\ell}}{2^{n_{c_j}}} - x_{6j\ell}\right) \cdot 2^{(m-\ell)} \geq 0$$

where the $2^{n}c_{j}$ in the denominator of the compute term guarantees that a node is not overlaid before it is completely computed.

3.6 I/O Contiguity Constraints

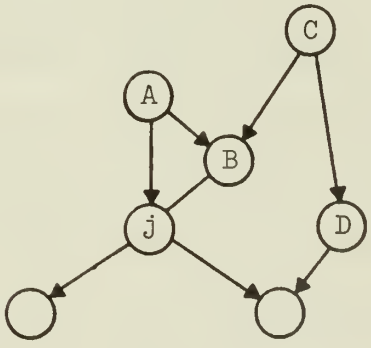
The constraints presented in this section perform two functions:

i) the structure of the code graph is represented here, and

ii) data which is needed for any calculation is forced to be entirely in primary memory before the calculation begins.

Definition:

The requirements of a node j (or, more simply, the requirements of j) are the predecessors of j. Thus, in the example:



A and B are the only requirements of j; C is a requirement of D, etc.

We require that for any node j, when $x_{ljk} = 1$, every one of the requirements of j be entirely in memory. [Clearly the requirements that the temporal order of execution in the solution be consistent with the code graph is met if the previous requirement is (data may not exist before it is calculated).]

$$\forall j, \forall k \quad x_{1jk} - \left[\sum_{\ell=0}^{k} \left(\frac{x_{1j\ell}}{n_{c_j}} + \frac{x_{3i\ell}}{p_i} - x_{6i\ell}\right]\right] \leq 0$$

for all i which are requirements of j^*

[The summation is seen to range between 0 and 1, as the portion of i in memory. The constraint simply guarantees that when j is being calculated ($x_{1jk} = 1$), the summation term must also be unity.]

3.7 System Constraints

It is here that constraints peculiar to the system being considered appear. Such items as multiprogramming, number and interrelation of data channels, etc., are "specified" as linear constraints.

It is not the purpose of this paper to anticipate the various system configurations one might consider. It is, rather, to indicate that appropriate constraints would be generated and to consider one specific example (albeit a simple one).

Consider a machine which can only transfer data over one channel at a time. This gives the constraints:

$$\sum_{j=1}^{n} (x_{3jk} + x_{5jk}) \leq 1$$

---

*Except for those nodes i which are initial nodes

This machine is further more not allowed to perform more than one "calculation" (as used in this paper) at a time:

$$\sum_{j=1}^{n} x_{1jk} \leq 1$$

There are assumed to be no further constraints, although it is obvious that many can be considered for various systems. Indeed, many differently configured systems may be described by suitably establishing the proper constraints here.

3.8 Criterion Function

Considering the constraints which require an overlay following a required output following a required computation for each node which is ultimately saved*, it suffices to further require only that the final (timewise) overlay be at the minimum time. That is

minimize $$\sum_{\ell=0}^{m} \left(\sum_{j=1}^{n} x_{4j\ell} \right) \cdot 2^{m}$$

*That is, ignoring those nodes which are "intermediate data" and never output.

4. BOUNDARY CONDITIONS

The constraints which have been given are for nodes which are computed. That is, nodes corresponding to replacement statements in the original code. It behooves us to now provide suitable alterations for these constraints as they apply to "initial nodes". (Initial nodes are those corresponding to input data and hence never get computed.)

Each initial node may be thought of as having the same set of six variables associated with it as do the other nodes. The first, x_{1jk}, is always 0 for $0 \leq k \leq m$, $1 \leq j \leq n$. This is done to allow as many of the previous constraints as possible to hold.

4.1 Constraints Which Apply to All Nodes

The following constraints are valid for initial nodes:

Memory constraints:
$$\sum_{r=0}^{k} \sum_{j=1}^{n} \left(\frac{x_{3jr}}{p_j} + \frac{x_{1jr}}{n_{c_j}} - x_{6jr} \right) S_j \leq Mem_{max}$$

(noting that the second term is always zero and may be omitted)

Disk timing constraints:
$$x_{2jk} - \sum_{t=s}^{f,r} x_{4jt} \leq 0$$

I/O contiguity constraints:
$$p_j \cdot x_{2jk} - \sum_{\ell=k}^{k+p_j-1} x_{3\ell k} \leq 0$$

$$p_j \cdot x_{4jk} - \sum_{\ell=k}^{k+p_j-1} x_{5\ell k} \leq 0$$

4.2 New Sequencing Constraints

It is necessary to change the sequencing constraints since initial nodes must have different temporal sequences of basic operations than the other nodes. This is due to the fact that

i) initial nodes are not computed;

ii) there is no reason to require a final copy on the disk as we assume that there is a copy there to begin with;

iii) since the initial copy is assumed to originate on the disk, it is not possible to reference its location there relative to its first (non-existent) output.

The allowable sequences, therefore, are

$$(IO^*D)\ (IO^*D)^*$$

Note that it will not be necessary to establish constraints requiring the initiation (IO*D) portion. Requiring (IO*D)* only is sufficient due to the fact that the in-memory constraints will force the input of the block at least once.

We first guarantee proper relative sequencing of I,D:

$$0 \leq \sum_{\ell=0}^{k} (x_{2j\ell} - x_{6j\ell}) \leq 0 \quad \text{forces} \quad (ID)^*$$

$$\sum_{\ell=1}^{k} (x_{2j\ell} - x_{6j\ell}) - x_{4jk+1} \geq 0$$

requires output to follow input (when it occurs). The remainder of the normal sequencing constraints are not applicable.

Note that the constraints in this section allow the initial nodes to be stored initially at any location on the disk, thus allowing full generality.

5. OTHER CONSIDERATIONS

5.1 Real Code

The method previously described is restricted to linear (sections of) code. In order to show some potential utility, however, it is necessary to show applications to "non-linear" code.

5.1.1 Looping

A loop may be thought of as concatenation of a section of linear code with itself as many times as the loop is executed. It is clearly impractical, though, to generate such "expanded code" and then proceed to optimize same. Another method will be given for "optimizing" loops which is at worst only slightly inefficient.

Consider the consequences of first optimizing a single iteration of a loop and then allowing the repetition (looping) of this "optimized" piece of code. The first time through the code would be linear and the execution time would indeed be minimal (in the sense discussed in this paper). Upon finishing execution of this first iteration, execution must commence at the "beginning" of the linear section of code.

In general the data needed to begin this execution will not be in an optimal location on the disk, thus requiring a wait due to "disk latency" of anywhere up to one revolution time. The remainder of this section of code will find items it needs from the disk at precisely those locations most advantageous. This is because after the initial wait, the computation of the second iteration of the loop is "in phase" with that of the first iteration. If the linear section (one complete iteration)

is of moderate length, it is reasonable to expect that the optimization process has reduced the running time by at least several disk revolutions (there would be little justification for it otherwise). This being the case, the loss of as much as only a single revolution per iteration is not outrageous--especially when we consider a latency of only one-half revolution on the average. The procedure of concatenating a previously optimized linear section, then, will still yield highly reduced running times over the original (non-altered) code, in spite of the fact that, in general, the execution of the loop is not really optimal.

Further refinements to the running time of loop may be obtained by concatenating the loop with itself some number of times (usually much less than the total number of iterations at run time). That a saving of time is effected is clear. As was previously explained, any loss of time due to disk latency (on the average, one-half a disk revolution) occurs when the optimized code is concatenated with itself. If this time loss can be made to occur once for every q iterations (as is the case when the original code is repeated q times and then optimized) instead of every iteration, the time lost is reduced by a factor of q in the long run.

This method would be particularly valuable for loops which take very little time to execute but will be repeated often. Such short loops need only be repeated several times before optimization.

5.1.2 Branching

In the absence of detailed information on the frequency and ordering of the various possibilities at each conditional transfer, it seems impossible to perform general inter-section optimization. Tentative

investigation in this area has not yielded any algorithms for obtaining an optimal solution.

It is, however, possible to improve on a brute-force section-by-section optimization. For example, in the case of a two-way conditional branch preceded by a section of linear code, the following "optimization" may be performed.

> The initial linear code and any one of the two branches are considered to be one section of linear code. (It may be necessary to add a dummy instruction at the branch point to account for the timing of the conditional test there.) Upon this "new" code is placed one additional restriction--while statements are allowed to be reordered for the normal optimization algorithm, reorderings over the boundary established at the branch point are prohibited. This may be easily done for example, through introduction of constraints requiring that every node in the initial section be computed before every node in the post-branch section.
>
> The second branch is considered by itself afterwards except that the initial conditions have already been determined by the previous optimization. In other words, the placement of data in primary and secondary memory at the branch point (resulting from the first optimization) are now the boundary conditions for the second branch and its optimization.

Several refinements of the above algorithm have been considered. Unfortunately, however, these may best be described as heuristic approaches since there is no real experience on which to base their value.

5.2 Overlapping Nodes

In many problems (e.g., PDE, weather codes) there are a set of basic operations which are carried out on an extremely large amount of data arranged on a grid. These operations require the values of points in local areas on the grid for each calculation at a point. Since it is necessary to partition these large grids into smaller sections, information must somehow be provided over these created boundaries. This can be done within the confines of the code lattice model as follows.

The requirements for each node include all over-boundary data (as many as 8 additional nodes for a five point finite difference method on a rectangular grid). Each calculation produces, besides the updated information for the section of grid, additional separate (although redundant) pieces of data--each of which is a requirement for at least one other node.

The previous constraints are perturbed such that while only one calculation is considered to have taken place, it produces a multiplicity of new nodes each of which is thereafter independent of its siblings.

5.3 Reduction of Variables

The procedure, as thus far described, yields 6 mn binary variables.* Besides generating a large number, this "brute force" method ignores

---

*Recalling, m = no. of time intervals, n = number of nodes.

information useful for reducing the number of variables. It is possible, though, to easily eliminate a large number of the variables.

Consider, for example, the node marked A' in the earlier example. Since the nodes A, B, C and D must all have been calculated before A' begins, it seems silly to consider the computation of A' "early" in the solution. Indeed, it is unnecessary to consider the compute variables (x_{1jk}) for all time intervals before the time in which it takes to compute all the predecessors for a given node. The required nodes are easily obtained by merely tracing up the graph (against the arrows) from the node under consideration; that is, all predecessors and their predecessors, recursively. It should also be clear that all variables (input, output, overlay) are forced to be 0 for these "early" time intervals in which the compute variables are also 0.

In a similar fashion, variables may be removed from "late" time intervals with two exceptions:

i) since no a priori knowledge of the optimal running time exists, the time intervals in question are those counting backwards from the maximum time; and,

ii) input, output, and overlay variables are not set to 0 as are compute variables in these regions.

There appears to be no analytic way of relating the saving (removing) of variables to the graph structure (which, indeed, it is highly dependent upon). Preliminary results indicate, however, a savings of the order of magnitude of one-half the total number of variables (6mn).

It should be noted that the programs for which there are great reductions of variables by the preceding process are those which have

code graphs which have many levels. Similarly, programs yielding code graphs which have few levels do not result in large reductions of variables. (Intuitively, long code graphs are those in which there are long chains of nodes dependent on their predecessors and hence many nodes which are constrained to start computing late--or finish computing early.)

APPENDIX A

SUMMARY OF CONSTRAINTS

Memory: $\sum_{r=0}^{k} \sum_{j=1}^{n} \left(\frac{x_{3jr}}{p_j} + \frac{x_{1jr}}{n_{c_j}} - x_{6jr}\right) \cdot (\text{Size of jth block}) \leq \text{Mem}_{max}$

Number of Constraints: m

Disk Timing: $x_{2jk} - \sum_{t=s}^{f,r} x_{4jt} \leq 0$ m·n

s = k modulo r, f = k-r

Continuous Transmission: $p_j \, x_{2jk} - \sum_{\ell=k}^{k+p_j-1} x_{3j\ell} \leq 0$ m·n

$p_j \, x_{4ik} - \sum_{\ell=k}^{k+p_j-1} x_{5j\ell} \leq 0$ m·n

Sequencing:

All Nodes: $\sum_{\ell=1}^{k} (x_{6j\ell} - x_{2j\ell}) \leq 1$ m·n

$\sum_{\ell=1}^{k} (x_{2j\ell} - x_{6j\ell}) \leq 0$ m·n

$\sum_{\ell=1}^{k} (x_{6j\ell} - x_{2j\ell}) + x_{4jk+1} \leq 1$ m·n

Non-Initial Nodes: $-\sum_{\ell=1}^{m} x_{6j\ell} \leq 1$ n

(standard) $\sum_{\ell=0}^{m} (x_{6j\ell} - x_{4j\ell}) \cdot 2^{(m-\ell)} \leq 0$

(temporary) $\sum_{\ell=0}^{m} (x_{6j\ell} - \frac{x_{1j\ell}}{2^{n_{c_j}}}) \cdot 2^{(m-\ell)} \leq 0$ } n

$$\sum_{\ell=0}^{m} x_{1j\ell} - n_{c_j} = 0 \qquad n$$

$$x_{1jk} + \sum_{\ell=0}^{k} x_{4j\ell} \leq 1 \qquad m \cdot n$$

Contiguity:

$$x_{1jk} - \sum_{\ell=0}^{k} \left(\frac{x_{1i\ell}}{n_{c_j}} + \frac{x_{3i\ell}}{p_i} - x_{6i\ell}\right) \leq 0 \qquad \sim 3m \cdot n$$

$\forall i$ which are requirements of j

System:

$$\sum_{j=1}^{n} x_{1jk} \leq 1 \qquad m$$

$$\sum_{j=1}^{n} (x_{3jk} + x_{5jk}) \leq 1 \qquad m$$

Total Number of Constraints: $\sim 10\ mn + 3(m+n)$

APPENDIX B

EXAMPLE

As an example of this method, let us consider the following sequence of replacement statements as a section of code which is to be optimized.

$$F = f_1(B,D)$$

$$J = f_2(A,F)$$

$$C = f_3(A,B)$$

$$L = f_4(C,J)$$

$$E = f_5(C,D)$$

$$G = f_6(B,E)$$

$$H = f_7(F,G)$$

$$K = f_8(F,H)$$

$$I = f_9(C,H)$$

where each of the "variables" A-L represents a block of data, their respective sizes being as follows:

(we will also here associate block numbers 1-12 for variables A-L, respectively; as well as normalizing the available primary memory to be unity)

Size block F = 1/8 memory therefore, size of the 6th block = 1/8

J = 1/16 memory therefore, size of the 10th block = 1/16

C = 1/4 memory therefore, size of the 3d block = 1/4

L = 1/4 memory therefore, size of the 12th block = 1/4

E = 1/4 memory therefore, size of the 5th block = 1/4

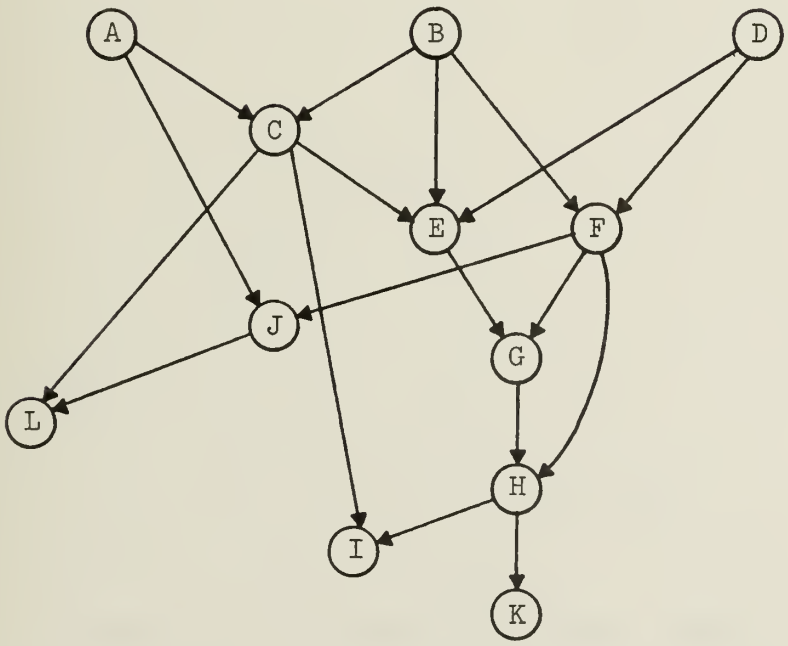
G = 1/8 memory therefore, size of the 7th block = 1/8

H = 1/8 memory therefore, size of the 8th block = 1/8

K = 1/16 memory therefore, size of the 11th block = 1/16

I = 1/16 memory therefore, size of the 9th block = 1/16

The code graph for this code is:



and we will assign the following time intervals necessary to evaluate each of the (arbitrary) functions f_1 - f_9.

Time for f_1, f_3, f_7, f_8, f_9 = 1 interval therefore, $n_{c_{1,\ldots,9}} = 1$

f_2, f_5, f_6 = 2 intervals therefore, $n_{c_{2,5,6}} = 2$

f_4 = 3 intervals therefore, $n_{c_4} = 3$

Where one revolution will be 4 intervals.

In an effort to keep this example non-complex, all blocks are assumed to be transmitted between primary memory and secondary memory in a single time interval ($p_j = 1,\ 1 \leq j \leq 12$).

We will further assume a maximum running time of 24 intervals ($m = 24$). As an example of the flexibility available in specifying constraints for particular problems, we will make two alterations to the constraints as previously described.

The first change deals with the constraints for initial conditions: it seems reasonable to consider that the initial nodes may be initially in memory (although it would not be reasonable to require that they do). This can be reflected by removing the constraint on the number of blocks which may be input during the 0th time interval. Thus, the solution may indicate that more than one block has been transmitted during the 0th time interval. This may in fact indicate that these blocks were loaded at the same time (just prior to run time) or may reflect the situation in which these blocks were input (one at a time) earlier (intuitively, during intervals -1,-2,...).

The second alteration is a minor one dealing with the use of memory. We will arbitrarily declare that memory can never be completely

used. That is, the inequality "$\leq$" in the memory constraint is replaced by "<" yielding:

$$\sum_{r=0}^{24} \sum_{j=1}^{12} \left(x_{3jr} - \frac{x_{1jr}}{n_{c_j}} - x_{6jr}\right) (S_j) < 1.0$$

Elimination of Variables

We will now consider dropping variables in cases where particular values are precluded because they are not physically possible.

For each of the non-initial nodes, we list those nodes which must be calculated first:

C	none
E	C
F	none
G	C, E, F
H	C, E, F, G
I	C, E, F, G, H
J	F
K	C, E, F, G, H
L	C, J

The earliest time interval in which any of these nodes may be calculated is one later than the sum of the time intervals it takes to calculate its predecessors. The following variables are therefore so constrainted.

$x_{i51} = 0 \qquad 1 \leq i \leq 6$

$x_{i7k} = 0 \qquad 1 \leq i \leq 6 \qquad 1 \leq k \leq 4$

$x_{i8k} = 0 \qquad 1 \leq i \leq 6 \qquad 1 \leq k \leq 6$

$x_{i9k} = 0 \qquad 1 \leq i \leq 6 \qquad 1 \leq k \leq 7$

$x_{i\ 10\ 1} = 0 \qquad 1 \leq i \leq 6$

$x_{i\ 11\ k} = 0 \qquad 1 \leq i \leq 6 \qquad 1 \leq k \leq 7$

$x_{i\ 12\ k} = 0 \qquad 1 \leq i \leq 6 \qquad 1 \leq k \leq 3$

(x_{ij0} is assumed = 0 $\quad 0 \leq i \leq 6$, $j = 3, 5 - 12$)

($x_{111} = x_{121} = x_{141} = 0$)

In a similar fashion, it is possible to eliminate variables from the "end" of the computation.

Node H must be computed before nodes I, K

Node G must be computed before nodes H, I, K

Node F must be computed before nodes G, H, I, J, K, L

Node E must be computed before nodes G, H, I, K

Node C must be computed before nodes E, G, H, I, K

thus

$x_{18k} = 0 \qquad 23 \leq k \leq 24$

$x_{17k} = 0 \qquad 22 \leq k \leq 24$

$x_{16k} = 0 \qquad 15 \leq k \leq 24$

$x_{15k} = 0 \qquad 20 \leq k \leq 24$

$x_{13k} = 0 \qquad 18 \leq k \leq 24$

The preceding optimization eliminates 258 variables from consideration.

The complete set of constraints for this example is listed below:

Memory:
$$\sum_{r=0}^{24} \sum_{j=1}^{12} \left(x_{3jr} - \frac{x_{1jr}}{n_{c_j}} - x_{6jr}\right)(S_j) \leq 1.0$$

$$S_1 = S_3 = S_5 = S_{12} = 0.2500$$

$$S_2 = S_4 = S_6 = S_7 = S_8 = 0.1250$$

$$S_9 = S_{10} = S_{11} = 0.0625$$

$$n_{c_1} = n_{c_3} = n_{c_1} = n_{c_8} = n_{c_9} = 1$$

$$n_{c_2} = n_{c_5} = n_{c_6} = 2$$

$$n_{c_4} = 3$$

Disk Timing:
$$x_{2jk} - \sum_{t=s}^{f,4} x_{4jt} \leq 0 \qquad 1 \leq j \leq n \qquad 0 \leq k \leq m$$

$$s = k \text{ modulo } 4, \qquad f = k-4$$

Initial Conditions:
$$x_{2jk} - x_{3jk} \leq 0$$

$$x_{4jk} - x_{5jk} \leq 0$$

$$\sum_{\ell=1}^{k} (x_{2j\ell} - x_{6j\ell}) \leq 1 \qquad j = 1, 2, 4 \qquad 0 \leq k \leq 24$$

$$\sum_{\ell=1}^{k} (x_{2j\ell} - x_{6j\ell}) \geq 0 \qquad j = 1, 2, 4 \qquad 0 \leq k \leq 24$$

$$\sum_{\ell=1}^{k} (x_{2j\ell} - x_{6j\ell}) - x_{4\,j\,k+1} \geq 0 \qquad j = 1, 2, 4 \qquad 0 \leq k \leq 24$$

Sequencing:

Non-Initial Nodes:

$$\sum_{\ell=1}^{k} (x_{6j\ell} - x_{2j\ell}) \leq 1 \qquad j = 3 \text{ and } 5 \leq j \leq n$$

$$\sum_{\ell=1}^{k} (x_{2j\ell} - x_{6j\ell}) \leq 0 \qquad j = 3 \text{ and } 5 \leq j \leq n \qquad 1 \leq k \leq 24$$

$$\sum_{\ell=1}^{k} (x_{6j\ell} - x_{2j\ell}) + x_{4\ j\ k+1} \leq 1 \qquad j = 3,5\text{-}n \qquad 1 \leq k \leq 24$$

$$\sum_{\ell=1}^{24} x_{6j\ell} \geq 1 \qquad j = 3,5\text{-}n$$

$$\sum_{\ell=0}^{24} x_{1j\ell} - n_{c_j} = 0 \qquad j = 3,5\text{-}n$$

$$x_{1jk} + \sum_{\ell=0}^{k} x_{4jk} \leq 1 \qquad j = 3,5\text{-}n \qquad 1 \leq k \leq 24$$

Contiguity:

$$x_{13k} - \sum_{\ell=0}^{k} (x_{31\ell} - x_{61\ell}) \leq 0 \qquad 1 \leq k \leq 24$$

$$x_{13k} - \sum_{\ell=0}^{k} (x_{32\ell} - x_{62\ell}) \leq 0 \qquad 1 \leq k \leq 24$$

$$x_{15k} - \sum_{\ell=0}^{k} \left(\frac{x_{1j\ell}}{n_{c_i}} + x_{3i\ell} - x_{6i\ell}\right) \leq 0 \qquad i = 2,3,4 \qquad 1 \leq k \leq 24$$

$$x_{16k} - \sum_{\ell=0}^{k} \left(\frac{x_{1j\ell}}{n_{c_i}} + x_{3i\ell} - x_{6i\ell}\right) \leq 0 \qquad i = 2,3 \qquad 1 \leq k \leq 24$$

$$x_{17k} - \sum_{\ell=0}^{k} \left(\frac{x_{1i\ell}}{n_{c_i}} + x_{3i\ell} - x_{6i\ell}\right) \leq 0 \qquad i = 2,5,6 \qquad 1 \leq k \leq 24$$

$$x_{18k} - \sum_{\ell=0}^{k} \left(\frac{x_{1i\ell}}{n_{c_i}} + x_{3i\ell} - x_{6i\ell}\right) \leq 0 \qquad i = 6,7 \qquad 1 \leq k \leq 24$$

$$x_{19k} - \sum_{\ell=0}^{k} \left(\frac{x_{1i\ell}}{n_{c_i}} + x_{3i\ell} - x_{6i\ell}\right) \leq 0 \qquad i = 3,6,8 \qquad 1 \leq k \leq 24$$

$$x_{1\ 10\ k} - \sum_{\ell=0}^{k} \left(\frac{x_{1i\ell}}{n_{c_i}} + x_{3i\ell} - x_{6i\ell}\right) \leq 0 \qquad i = 1,6 \qquad 1 \leq k \leq 24$$

$$x_{1\ 11\ k} - \sum_{\ell=0}^{k} \left(\frac{x_{1i\ell}}{n_{c_i}} + x_{3i\ell} - x_{6i\ell}\right) \leq 0 \qquad i = 6,8 \qquad 1 \leq k \leq 24$$

$$x_{1\ 12\ k} - \sum_{\ell=0}^{k} \left(\frac{x_{1i\ell}}{n_{c_i}} + x_{3i\ell} - x_{6i\ell}\right) \leq 0 \qquad i = 3,10 \qquad 1 \leq k \leq 24$$

System:

$$\sum_{j=1}^{12} x_{1jk} \leq 1 \qquad 0 \leq k \leq 24$$

$$\sum_{j=1}^{12} x_{3jk} + x_{5jk}) \leq 1 \qquad 1 \leq k \leq 24$$

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)	2a. REPORT SECURITY CLASSIFICATION
Department of Computer Science University of Illinois at Urbana-Champaign Urbana, Illinois 61801	UNCLASSIFIED
	2b. GROUP

3. REPORT TITLE

A MODEL FOR LINEAR PROGRAMMING OPTIMIZATION OF I/O-Bound Programs

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Research Report

5. AUTHOR(S) (First name, middle initial, last name)

David E. Gold

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
June, 1969	43	0

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
46-26-15-305 b. PROJECT NO. USAF 30(602)4144 c. d.	DCS Report No. 340 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

10. DISTRIBUTION STATEMENT

Qualified requesters may obtain copies of this report from DCS.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
NONE	Rome Air Development Center Griffiss Air Force Base Rome, New York 13440

13. ABSTRACT

A model of a class of machines having periodically addressable secondary memory is derived. The problem of (timewise) optimization of programs consisting of many core-loads is considered with respect to the model. For a large class of programs, sufficient constraints are established to make the solution amenable to linear programming methods.

DD FORM 1 NOV 68 1473

UNCLASSIFIED

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
ILLIAC IV						
input output						
optimize computation						
computation model						
disk timing						
memory utilization						
linear programming						

JAN 29 1973

UNIVERSITY OF ILLINOIS-URBANA
3 0112 056690867